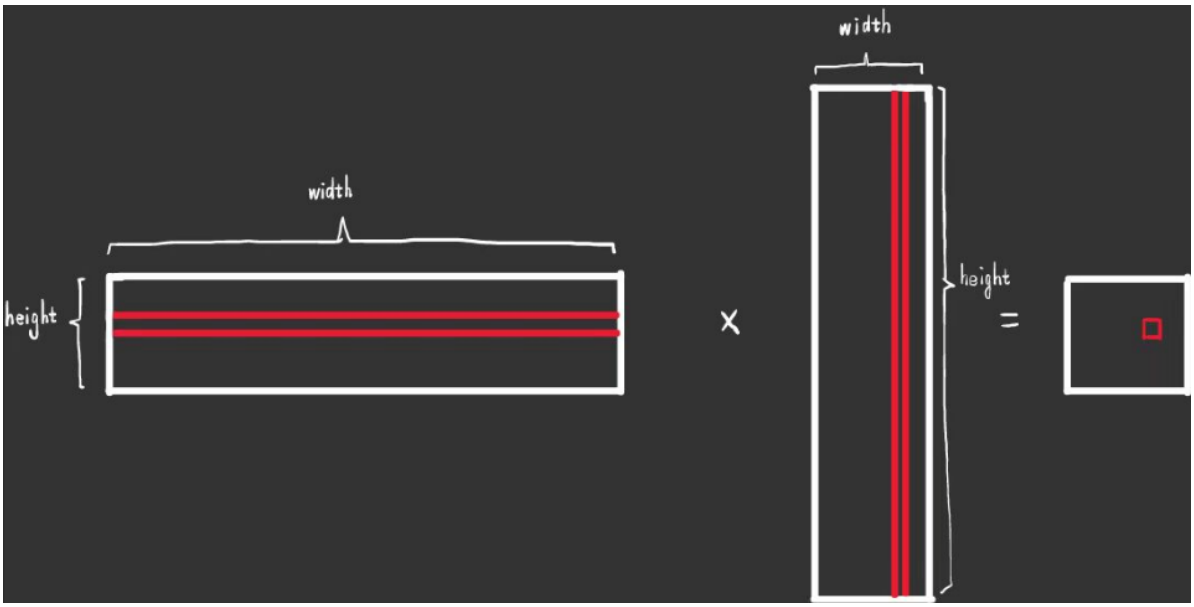
Project —— Matrix Multiplication with CUDA

1. Tile size 1×1 per thread

在本方法中，每一个thread负责抽取A中的一行和B中的一列进行计算（即一个tile），得到C中的一个元素，如图所示：



注意到矩阵是以row-major的方式储存的，代码如下：

```
int sum = 0;
for (int i = 0; i < A.width; i++)
{
    sum += A.elements[idx_y * A.width + i] * B.elements[i * B.width + idx_x];
}
C.elements[idx_y * C.width + idx_x] = sum;
```

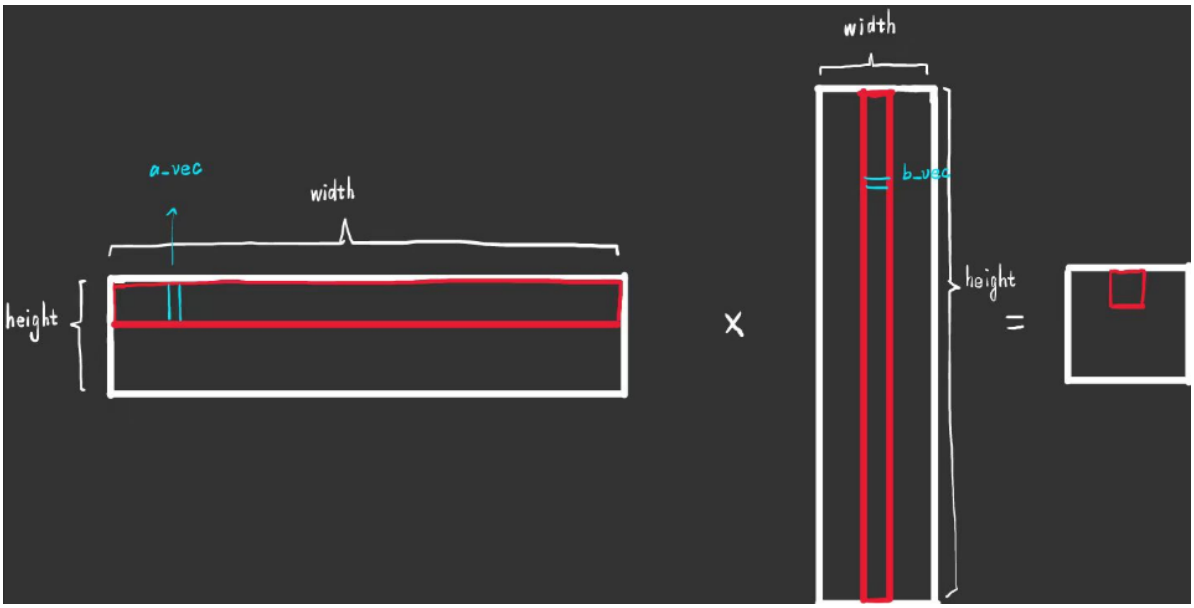
运行结果如下：

```
[tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication
CUDA Elapsed time: 3.274912 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication
CUDA Elapsed time: 2.857984 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication
CUDA Elapsed time: 3.019360 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication
CUDA Elapsed time: 2.876256 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication
CUDA Elapsed time: 3.003520 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$
```

该方法的平均运行时间为：3.006406 ms。

2. Incresing tile size per thread

本方法中，每一个thread负责的tile数增多，以增强数据的复用性。如图所示：



首先对`Csum`中的元素清零：

```
memset(Csum, 0, sizeof(Csum));
```

之后初始化`a_vec`和`b_vec`，并做乘累加运算：

```
for (int i = 0; i < TILE_SIZE; i++)
{
    a_vec[i] = A.elements[(tile_row + i) * A.width + k];
    b_vec[i] = B.elements[k * B.width + (tile_col + i)];
}

for (int i = 0; i < TILE_SIZE; i++)
{
    for (int j = 0; j < TILE_SIZE; j++)
    {
        Csum[i][j] += a_vec[i] * b_vec[j];
    }
}
```

循环结束之后，将`Csum`中的元素进行写回：

```
for (int i = 0; i < TILE_SIZE; i++)
{
    for (int j = 0; j < TILE_SIZE; j++)
    {
        C.elements[(tile_row + i) * C.width + (tile_col + j)] = Csum[i][j];
    }
}
```

运行结果如下(`TILE_SIZE=2`)：

```
[tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication_method2
CUDA Elapsed time: 2.205408 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication_method2
CUDA Elapsed time: 2.335296 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication_method2
CUDA Elapsed time: 2.326528 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication_method2
CUDA Elapsed time: 2.232832 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication_method2
CUDA Elapsed time: 2.255392 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$
```

改变`TILE_SIZE`的大小，并对运行时间进行统计：

tile size	time(ms)
1	3.282611
2	2.271091
4	2.107053
8	2.077536
16	7.368474
32	25.815271
64	83.601498

随`TILE_SIZE`的增加，运行时间先减小后增大（在8左右达到最小值，随后急剧增大）。

1. 注意到此处A和B均存储在global memory中，访存开销较大。当tile size从1增加到8时，对A中特定行/B中特定列的访存数减小，计算密度（MACS操作的数目与访存操作的比例）增大；
2. 考虑到每一个warp中含有32个线程，最多含有256个向量寄存器，其中每个寄存器可以存放32个32位元素，均摊在每一个线程上，最多能有256个元素能被存放于寄存器中。而当tile size大于16时，光`C_sum`一项中就含有256个32位int元素，多余的元素只能存放在L1 cache中，造成较大的开销。此外寄存器使用率接近100%时，可能会造成active warp数减小、bank conflict等。
3. 本服务器的GPU有56个SM。当tile size变为16时，block num变成了16（相比之下，当当tile size为8时，block num变成了64），没有充分利用GPU内的SM，并行度不够。

> 一个小疑问：不同`grid`之间的线程真的是完全串行执行的吗？
>
> 首先，一个block中的所有线程必须在一个SM上进行操作(但并不是说一个SM上同时只能处理一个block中的数据——SM一次只会并行执行多个block中的一个warp，但不是一定执行完毕，当某个块中的warp在存取数据时，会切换到同一个块中的其他warp执行)，必须共享所有该内核的资源。毫无疑问，一个block中的线程一定是并行的。但是grid中的block是独立执行的，多个block可以采用任何顺序执行操作，即并行，随机或顺序执行。CUDA在运行kernel时，会将线程在SM层面上划分为若干组，每组共32个thread，成为warps。在处理完某个block中的所有thread后，SM会找还没有处理的block进行处理。
>
> 要想实际衡量grid之间的并行能力，或者说，看看GPU上的SM是否有闲置，需要了解GPU中实际的SM数量。假设一个任务有5个block、GPU有3个SM， 一个SM最多可以同时算1个block。那么，5个block需要2波（5 / 3）才能算完（如果SM上有执行block的切换，那么可以将其理解为使用了所谓“时间片”机制，时间会有所延长，平均下来，其耗时表现与同时算1个block差不多，可能甚至更

> 大，此处不作过细考虑）。
> 编写以下测试程序：

```
#include <iostream>
int main(int argc, char **argv)
{
int dev = 0;
cudaDeviceProp devProp;
cudaGetDeviceProperties(&devProp, dev);
std::cout << "GPU device " << dev << ": " << devProp.name << std::endl;
std::cout << "SM number: " << devProp.multiProcessorCount << std::endl;
std::cout << "shared memory per block: " << devProp.sharedMemPerBlock /
1024.0 << " KB" << std::endl;
std::cout << "max threads per block: " << devProp.maxThreadsPerBlock <<
std::endl;
std::cout << "max threads per multiprocessor: " <<
devProp.maxThreadsPerMultiProcessor << std::endl;
}
```

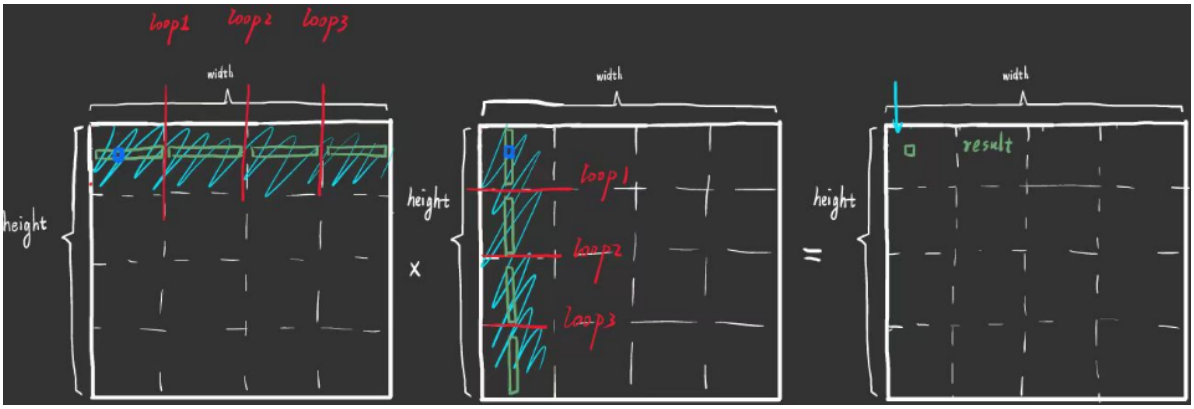
> 得到本服务器的数据如下：

```
GPU device 0: Tesla P100-SXM2-16GB
SM number: 56
shared memory per block: 48 KB
max threads per block: 1024
max threads per multiprocessor: 2048
```

> 可见有56个SM，执行任务中的block num数肯定超过了这个数。可见不存在未被利用的SM，并行性已经发挥到了极限。

3. Optimization using shared memory

注意到在上例中，一段重复的代码可能会被多个thread所用到，故考虑将其载入shared memory中。如图所示：



如图所示，深蓝色部分表示每一个线程在每一次循环时需要载入的数据。一个block内的线程在载入过后，需要调用一次`__syncthreads()`来确保全部载入完毕。随后，绿色部分表示每一个线程覆盖的计算部分。浅蓝色是每次loop时，需要载入shared memory的数据。

首先在计算之前，要确保shared memory正确初始化：

```
As[row][col] = GetElement(Asub, row, col);
Bs[row][col] = GetElement(Bsub, row, col);
__syncthreads();
```

之后从shared memory中读取数据进行乘累加运算：

```
for (int e = 0; e < BLOCK_SIZE; ++e)
{
    Cvalue += As[row][e] * Bs[e][col];
}
__syncthreads();
```

最后写回结果：

```
SetElement(Csub, row, col, Cvalue);
```

运行结果如下(`TILE_SIZE=16`)：

```
[tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication_opt
CUDA Elapsed time: 1.872832 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication_opt
CUDA Elapsed time: 1.873216 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication_opt
CUDA Elapsed time: 1.858048 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication_opt
CUDA Elapsed time: 1.824800 ms
Success!![tsinghuaee273@i-isw0lefk cuda_project]$ ./matrix_multiplication_opt
CUDA Elapsed time: 1.820800 ms
```

改变`TILE_SIZE`的大小，并对运行时间进行统计：

tile size	time(ms)
1	62.395489
2	10.850624
4	2.765997
8	2.107232
16	1.849939
32	1.928793

首先，引入shared memory之后，每一个thread无需像上一种方法那样，维护自己的`C_sum`数组，这样寄存器的使用状况得到了缓解；当tile size增大时，数据的复用性增强，只要不超过线程块所支持的最大线程数，执行时间就会一直减小。

Reference